Backward and forward compatibility

Suresh Podeti

3 min read · May 21, 2023

Mostly, in relational databases we need to define **database schema** so that applications can write to database, and read from database. All the data in the database conforms to **one schema**: although that schema can be changed.

For any application, adding **new feature** or **updating existing feature** requires **schema changes**. Changing schema is called **schema evolution**. Changing a schema may involve adding a new field, updating, and deleting an existing field.

In case of, schema-on-read (schema-less) eg. NoSQL documents, databases don't enforce a schema, such database can contain a **mixture of older and newer data formats** written at different times.

When **schema changes**, often applications need to **change code** to use changed schema to write and read from database. In large applications change in code cannot happen instantaneously:

1. **Server-side applications** — We should perform a **rolling update**, deploying the new version of code to a few nodes at a time, checking whether the new version is running smoothly, and gradually working our way through all the nodes.

2. **Client-side applications** — we are at the mercy of the users, who may not install the update for some time.

Old and new versions of the code, and old and new data formats may all **coexist** in the system at the same time. In order for the system to continue running smoothly, we need to maintain compatibility in both directions:

Backward compatibility

Newer code can read data that was written by older code.

Backward compatibility is normally not hard to achieve: as author of the newer code, we know the format of data written by older code, and so we can explicitly handle it (if necessary by simply keeping the old code to read the old data).

Forward compatibility

Older code can read data that was written by newer code

Forward compatibility can be trickier, because it requires older code to ignore additions made by a newer version of the code.

We should aim to build systems that make it easy to adapt change (and also called as **evolvability**).

Analogy

Consider an example from pizza parcel outlet. *Pizzaiolo* a chef who makes and packs pizza into a box. Assume, current Pizza version: **Pizza v1.0**, and box version: **Box v1.0.**

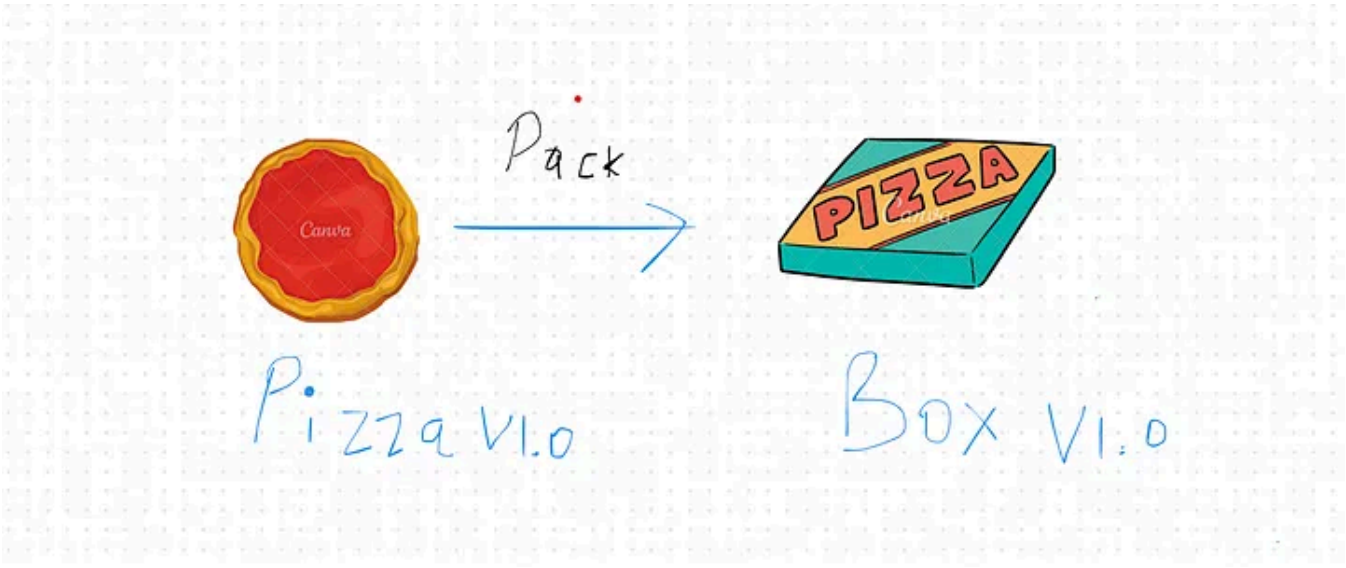

Fig 1.0: An example to illustrate backward and forward compatibility

Let say outlet rebranded their box by changing the colour, and box size (size greater than previous version) and say to version *Box v2.0*. Newer version of Box (*Box v2.0*) should able to accommodate old version of Pizza (Pizza v1.0) so it is said be **backward compatible.**

If Box v2.0 is smaller than the previous version Box v1.0 then it won't fit Pizza v1.0 into the box so it won't be backward compatible.

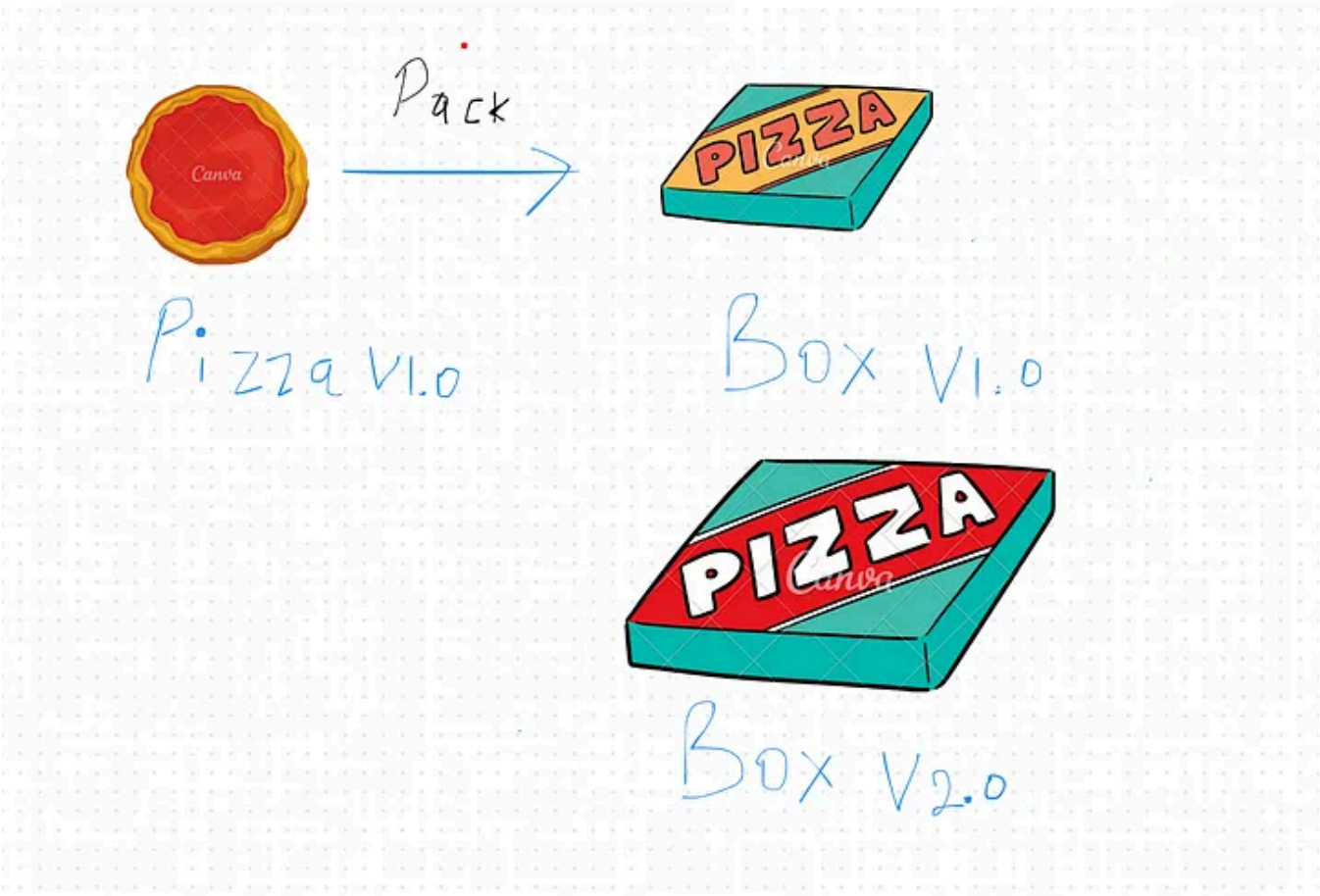

Fig 2.0: Box is updated to Box v2.0 by rebranding

Now, consider instead of updating Box v1.0 to Box 2.0, Let's say we want to change size of pizza (decrease pizza size — Pizza v2.0). Older version of box (Box v1.0) can fit newer version of pizza (Pizza v2.0) so it is said to be **forward compatible.**

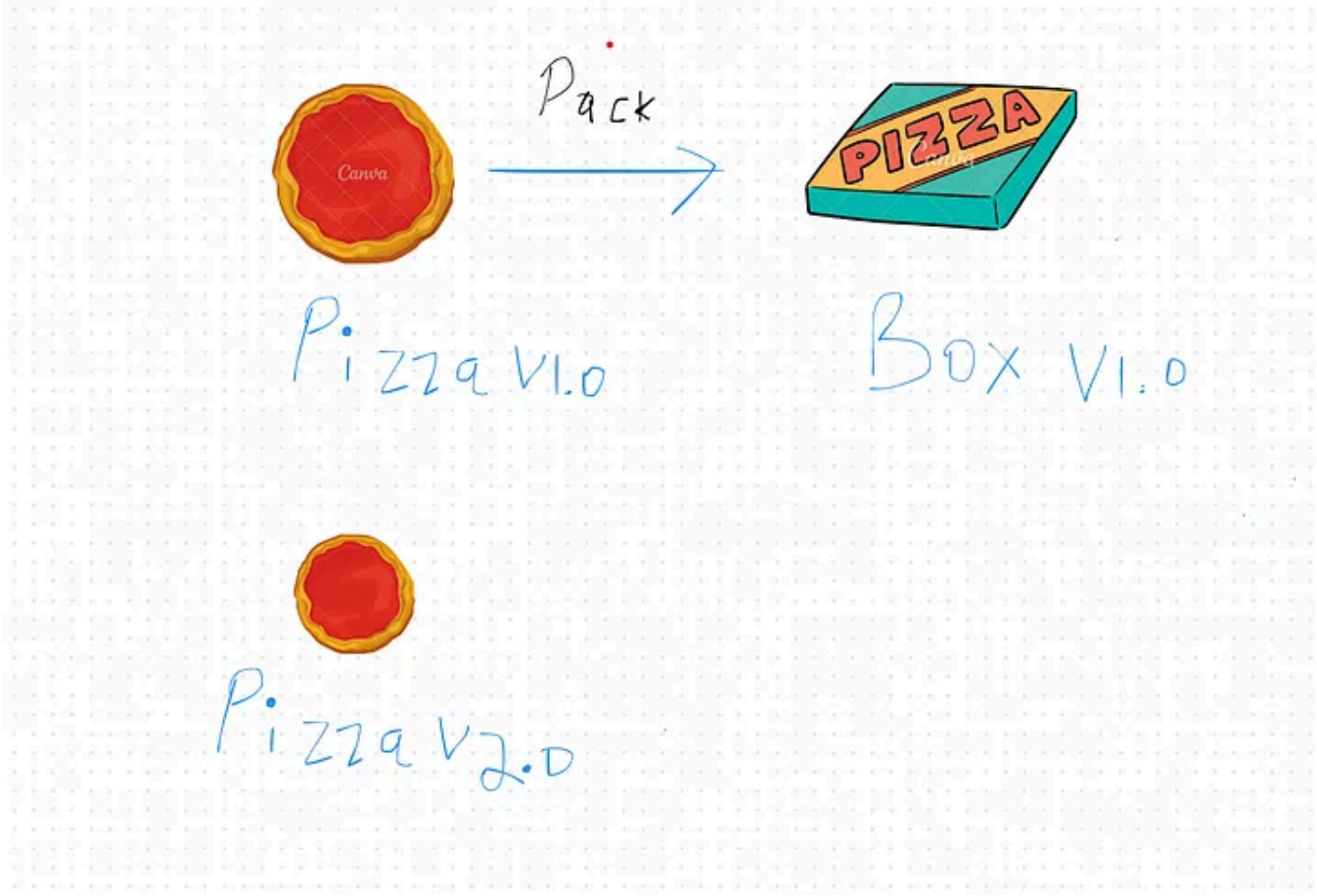

Fig 3.0: Pizza v1.0 updated to Pizza v2.0 by decreasing the size

We can map our example as follows:

Code — *Pizzaiolo*, and Box

Data — Pizza

Packing Pizza — equivalent to read

Making pizza — equivalent to write

Reference:

1. O'Reilly designing data-intensive applications by Martin Kleppmann, Chapter 4: Encoding and Evolution

Compatibility | Backwards Compatibility | Forward Compatibility | Rolling Updates | Evolution

Edit profile

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience

Recommended from Medium